UNIX Threads

A study of threads in Unix and how to use them

Pratap Vikram Singh

November 5th, 1996

Contents

1. Threads in UNIX

This section gives an overview of the theory behind threads in UNIX.

1.1 Introduction to Threads

A *thread* is simply an execution stream through a process. In the traditional UNIX process model, a single thread of execution starts with the first instruction of the function `main()` and follows the logic of the application till the process terminates. The traditional process model incorporates the address space of the process such that every process executes in a disjoint address space.

With the UNIX thread interface, an application now has one or more execution streams within the same process. Each execution stream is a *thread of execution*, or simply a thread. A UNIX process with a single thread can create additional threads within the same process address space.

A thread separates a process's sequential execution stream from its other resources. The resources are shared between the multiple execution streams. The threads execute as concurrent execution streams sharing the same address space performing tasks associated with the desired services.

To support separate execution streams, each thread requires a piece of executable code with a stack and data. The user level stack provides for per-thread local variable administration. From the OS perspective, each of the executing threads must have a separate kernel stack and process context. This will allow each thread to execute system calls, and to service interrupts and page faults independently.

Collectively, the threads executing in the same address space are called *sibling* threads. The term is not same as a parent and child relationship but more of a peer relationship.

1.2 Thread Implementations

There are three types of thread implementations:

1. Kernel level abstraction.
2. User level abstraction.
3. Mix of both.

Kernel supported threads requires kernel data structures such as a process table entry and a user table entry to represent the thread. These threads are basically processes and require direct support from the OS. In the kernel supported thread model, each thread has its own context, including the private storage area, volatile registers and a stack. A kernel supported threads model has the disadvantage that the kernel size will increase with each thread added to any process.

User supported threads are represented by data structures within a process's own address space. These threads don't require direct support from the OS. A user level thread has the potential to execute only when associated with a kernel process. The implementation must therefore *multiplex* a user level thread on to a kernel process for execution and later preempt it in favor of running a sibling thread. The advantage of this implementation is that this does not require the process to change its mode from user to kernel in order to schedule multiplexed threads. This can permit thread creation, scheduling and eventual termination to complete with better performance than an equivalent kernel supported thread implementation.

A process can use one kernel thread per user level thread. The kernel thread is a *light weight process* (LWP). A LWP can be thought of as a virtual CPU which is available for executing code or system calls. Each LWP is dispatched separately by the kernel, may perform independent system calls, incur independent page faults and may also run in parallel on a multiprocessor architecture. The user level thread is bound to the kernel LWP. System scheduling, dispatching and execution of kernel LWPs will result in the execution of the *bound* user level thread. The disadvantage of bound user threads is that each user level thread requires the creation of a kernel LWP. This consumes the kernel processing time. Similarly, when the user level thread blocks then the kernel LWP also blocks and cannot do anything else.

1.3 Thread Bindings

The UNIX threads interface provides for *multiplexed* and *bound* threads. *Multiplexed* threads are a user level abstraction and *bound* threads have to be directly supported by the kernel.

The default thread creation assumes a multiplexed thread and allocates a data structure in the process's user address space to represent the newly created thread. User level scheduling of the thread will map it to an available kernel LWP for execution. The OS will schedule the kernel LWP based on the characteristics of the mapped thread. Some implementations maintain a group of LWPs for this purpose called, a *thread pool*.

With multiplexed threads implemented at the user level, the UNIX process can have more threads than the available kernel LWPs.

1.4 Thread and kernel LWP bindings

THREADS	LWPs	BINDING RELATIONSHIP
1	1	1 to 1 (Bound)
M	1	M to 1
1	N	(N-1) Unused
M	N	M to N

Attributes can be applied to describe characteristics of a thread. Both *multiplexed* and *bound* threads can have attributes. The two attributes currently supported are *daemon* and *detached* threads.

The *daemon* thread attribute modifies, how a process termination occurs. A UNIX process will terminate when the last thread exits or the process voluntarily terminates by calling the `exit()` function. The *daemon* attribute affects the first situation and permits the UNIX process to terminate when the last *non-daemon* thread exits. When this happens, the UNIX process will silently terminate the *daemon* threads.

The *detached* thread attribute modifies the availability of the thread's termination status. It indicates that the application program does not require the exit status of the detached thread.

1.5 Thread Resources

Sibling threads can share certain resources associated with the UNIX process like global memory, file descriptors and process identifier. Sibling threads executing in the same address space are not defined any boundary protection among themselves. When a thread modifies the value of a global variable, the new value is immediately visible to all the siblings. Also if one thread issues a `chdir()` function, all threads effectively change their current working directory.

1.6 Thread Private Data

Resources that cannot be shared by sibling threads are called *thread private data*. For example, each thread has a unique identifier in a process. The thread private data is determined by the implementation.

Shared Resources	Private Resources
Global Memory	Thread Identifier
Current Working Directory	Interval Timers
User Identification	Signal Mask
File Descriptors	Registers
Shell Environment	Errno
Umask	Instruction Pointer

1.7 Thread Specific data

Sibling threads can have their own view of data items identified as *thread specific data*. A UNIX process's threads should not share application specific error variables.

Thread specific data is different from thread private data. The UNIX threads implementation defines the thread private data, while the application defines the thread specific data.

For example an application designed with multiple threads can use a thread specific global error variable to report error conditions.

1.8 Thread Execution

A process's threads execute in a single UNIX process environment. This environment holds all the resources shared by the sibling threads. The thread execution environment describes the scheduling policy and the priority value for a thread.

From the OS point of view, the process has one or more kernel LWPs that provide the execution for the threads. The threads are either multiplexed to these LWPs or are bound to them.

Multiplexed threads can execute only when mapped on to a kernel LWP. The mapping service will propagate the characteristics of the thread to the kernel LWP. The OS uses these characteristics to schedule and subsequently dispatch the kernel LWP for execution.

When the kernel LWP issues a kernel call on behalf of the multiplexed thread, the thread remains bound to the LWP until the kernel completes the call. If the kernel call blocks, the multiplexed thread along with its kernel LWP will also block.

If the kernel LWP from the pool is preempted by the OS or it voluntarily gives up the processor, the state of the thread is saved in the UNIX process's user address space. A new thread can then be mapped to the kernel LWP.

There is, in general, no way to predict how the instructions of different threads are interleaved.

1.9 Steps in thread execution

1. An LWP chooses a thread to run by locating the thread state in process memory.
2. After loading the registers and assuming the identity of the thread, the LWP executes the thread's instructions.
3. If the thread cannot continue, or if other threads should be run, the LWP saves the state of the thread back in memory.
4. The LWP can now select another thread to run.

1.10 State of a thread

The state of a thread is defined by the following :

- Thread ID.
- Register State (Including the instruction pointer and the stack pointer).
- Stack.
- Signal Mask.
- Priority.
- Thread local storage.

1.11 Thread Scheduling

The threads library implements a thread scheduler that multiplexes thread execution across a pool of LWPs. This allows any thread to execute on any of the LWPs in the pool. When a thread executes, it is attached to a LWP and has all the attributes of being a kernel supported thread.

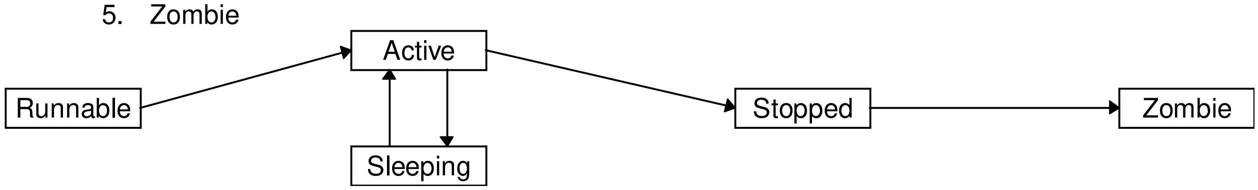
All runnable, unbound threads are on a user level, prioritized dispatch queue. Thread priorities range from 0 to infinity *(the maximum number representable by 32 bits)*. A thread's priority can be changed only by the thread itself or by another thread in the same process. The priority of an unbound thread is known only to the user level scheduler and not to the kernel.

An LWP in the pool is either *idling* or *running* a thread. When a thread is idle, it waits on a synchronization variable. When a thread is made *runnable*, it is added to a *dispatch queue* and an idle LWP from the pool is awakened by signaling the synchronization variable. The LWP after waking up, switches to the highest priority thread on the dispatch queue. In the course of its execution, if the running thread blocks on a synchronization variable, e.g. *mutex lock*, the running LWP puts it in the sleep queue and switches to the highest priority thread in the dispatch queue. If the dispatch queue is empty, the LWP goes back to its *idle* state in the LWP pool. Threads in the sleep queue become runnable when their synchronization locks are freed. When a thread becomes *runnable,* it is put back in the dispatch queue. If all the LWPs in the pool are busy when a thread becomes *runnable*, it remains on the dispatch queue waiting for an LWP to become available. An LWP is made available either when a new one is added to the pool or when one of the running threads blocks exits or is preempted.

1.12 Thread states

An unbound thread can be in one of the five different states :

1. Runnable
2. Active
3. Sleeping
4. Stopped
5. Zombie



1.13 LWP states

An LWP can be in one of the following four states :

1. Running
2. Stopped
3. Blocked
4. Runnable

1.14 Important scenarios

When an unbound thread exits and there are no more *runnable* threads, the LWP that was running the thread switches to a small idle stack associated with each LWP and idles by waiting on a global LWP condition variable. When another thread becomes *runnable*, the global condition variable is signaled, and the *idling* LWP wakes up and attempts to run any *runnable* threads.

When a bound thread blocks on a process local synchronization variable, its LWP must also stop running. It does so by waiting on a LWP semaphore associated with the thread. The LWP is now *parked*. When the bound thread unblocks, the parking semaphore is signaled so that the LWP can continue executing the thread.

When an unbound thread becomes blocked and there are no more *runnable* threads, the LWP that was running the thread also *parks* itself on the thread's semaphore rather than *idling* on the idle stack and the global condition variable. This is done to optimize the case where the blocked thread becomes *runnable* quickly and thus avoiding the context switch to idle stack and back to the same thread.

1.15 Threads Library Implementation

Each thread is represented by a thread structure that contains a thread identifier, an area to save the thread execution context, the thread signal mask, the thread priority and the pointer to the thread stack. The storage for the stack is either automatically allocated by the library or it is passed in by the application on thread creation. Library allocated stacks are obtained by mapping in pages of *anonymous* memory. Due to the growing nature of the stack, the library ensures that the page following the stack is invalid. This represents a *red zone* so that the process will be signaled if the thread should run off the stack. If the application passed its own stack, it can provide a red zone or pack the stacks in its own way.

When a thread is created, a thread ID is assigned. The TID is used as an index in a table of pointers to thread structures. This allows the library to give meaningful errors to each thread.

2. Using Threads

This section describes the system calls for using threads in UNIX.

2.1 Ceating a thread

A thread of a UNIX process can create a sibling thread by calling the `thr_create()` function.

2.1.1 Syntax

```
#include <thread.h>
int thr_create(
      void        *stack_address,   /* Stack area */
      size_t      stack_size,       /* Size of stack area */
      void        *(start_routine)(void *arg),  /*  Point  of  thread  execution
*/
      void        *arg,             /* Pointer to argument list */
      long         flags,           /* Execution and scheduling flags */
      thread_t    *new_thread);     /* Thread identifier */
```

The first parameter is the address of the stack area. In calling the `thr_create()` function, the invoking thread must specify a user level stack for the newly created thread. The `stack_address` parameter identifies the stack area. The calling thread can request the `thr_create()` function to provide a default user level stack or can specify a stack defined by the application. For most applications, the default stack will suffice and will sometimes give better performance because it is maintained by the thread library. By calling the `thr_create()` function with 'NULL' will tell the thread library to allocate a default stack for the new thread. The initialization is done by the library. When using a predefined stack, the application must allocate sufficient memory for the new stack. The `stack_address` pointer must point to this allocated block of memory. The thread library will initialize this allocated area and use it for the stack for the newly created thread.

The size of the thread stack is specified by the *`stack_size`* parameter. A stack size of zero will result in a default stack size determined by the thread library. A specific stack size can be requested by specifying the *`stack_size`* as a non-zero value. This value is used by the library to initialize the stack. When using a user defined stack, the *`stack_size`* specifies the number of bytes allocated. In specifying the size of the stack, *`stack_size`* must be greater than or equal to the minimum stack size required. If not so, the *`thr_create()`* function will fail and return with an *EINVAL* error condition.

2.1.2 Relationship between `stack_address` and `stack_size`

STACK ADDRESS	STACK SIZE	RESULTS
NULL	0	A default stack allocation will be used and the stack will have a default size.
NULL	> Minimum Stack size	A default stack allocation will be used to create a stack with the minimum size required by the implementation.
<ADDRESS>	0, >= Minimum Stack Size	The stack for the new thread begins at the specified address and is greater than or equal to the minimum stack size required by the implementation.

The created thread will begin execution with the function specified as *`start_routine`*. The function has to be declared as a *`void *<function name>(void *argumentlist)`*. The return type of the function must be of the type *`void *`* and the parameter list must be *`void *`* list of arguments. This is necessary because this is how the thread library defines the function interface.

The *`void *arg`* is the pointer to the list of arguments which have to be passed on to the newly created thread. It can be any kind of pointer from a string to an array. It is directly accessible from the new thread and can be deciphered as wished by the internal code of the thread. Because of the pointer being of type *void*, it may have to be type casted before use.

The default action of the *`thr_create()`* function is to create a multiplexed thread and to transition the thread to a *runnable* state. This default behavior can be modified through the *`flags`* parameter, which is a bitwise *OR* of zero or more of the available flags shown in the above table.

2.1.3 Binding relationship flags for `thr_create()`

THR_SUSPENDED	When the thread is created, it enters the RUNNABLE state by default. With this flag set, the new thread will enter the SUSPENDED state. The thread will not begin execution till an explicit call to `thr_continue()` is made by a sibling thread. Creating a thread in this mode allows the application to modify the scheduling parameters and other resources associated with the new thread prior to the execution of the `start_routine`.
THR_DETACHED	The exit status of the created thread will not be available to the sibling threads. This permits the UNIX thread library to reuse all of the thread's resources immediately after it terminates.
THR_NEW_LWP	In addition to creating a new thread, a new LWP is to be created and added to the pool of available LWPs. The `thr_create()` function does not create an LWP but merely requests an extra LWP in the LWP pool. The request may never be fulfilled.
THR_BOUND	The thread is bound to a new kernel LWP. Both, the thread and the kernel LWP, will be created as a result of the call. The thread will be bound only to this LWP and cannot execute on another LWPs. Similarly, this LWP will execute only this thread and not any other. The thread is said to be permanently bound to the LWP.
THR_DAEMON	The created thread is a daemon thread. A UNIX process will not exit until all non-daemon threads have exited, an explicit `exit()` has been called or the initial thread completes without calling `thr_exit()`. The UNIX thread library will cause the process to exit if the only remaining threads are daemon threads.

When a new has successfully been created, its thread identifier will be stored in the location pointed at by the `new_thread` parameter. If the `new_thread` parameter is specified as a NULL value, then the new thread identifier is not returned to the calling thread.

2.1.4 Return Value

Zero is returned when successful. A non-zero value indicates an error. If any of the following conditions are detected, *thr_create* ails and returns the corresponding values shown the following table.

EAGAIN	A system limit is exceeded, e.g. too many LWPs were created
ENOMEM	Not enough memory was available to create the new thread.
EINVAL	`stack_base` is not NULL and `stack_size` is less than the value returned by `thr_min_stack()`.
EINVAL	`stack_base` is NULL and `stack_size` is not zero and is less than the value returned by `thr_min_stack()`.

2.1.5 Sample Code

This example shows the different ways to create a thread:

```
#include <thread.h>
#include <stdio.h>
void *function(void *arg);
int main()
{
      void *address;
      int error;
      thread_t id;
      char string[]="Hello World";

      /* Create thread using default stack and size */
      error = thr_create(NULL, 0, function, NULL, 0, NULL);

      if( error != 0 ) return;
      address=(void *) malloc(1, 2048);

      /* Create thread using defined stack and size */
      error = thr_create(address, 2048, function, NULL, 0, NULL);

      if( error != 0 ) return;

      /* Create a bound thread and request for a new LWP. Pass the string str
       * as a parameter and get the thread      identifier in id.
       */
      thr_create(NULL, 0, function, string, THR_BOUND | THR_NEW_LWP, &id);

      printf("The thread id is: %d\n", id);
}

void *function(void *arg)
{
      /* Function Body */
}
```

2.2 Determining the minimum stack requirements

The function `thr_min_stack()` determines the minimum stack size required by a thread function. The minimum stack size is influenced by the size of the internal thread data structures and the thread private data required by the application.

2.2.1 Syntax

```
#include <thread.h>
size_t thr_min_stack(void);
```

For most multithreaded applications, the default stack size provided by the implementation will suffice. Certain applications might require a larger stack and must specify the stack size as *thr_min_stack* plus the application specific size E.g.: 2K.

2.2.2 Sample Code

```
/* Rest of code */
thr_create(NULL, thr_min_stack() + 2048, function, NULL, 0, NULL);
/* Rest of code */
```

2.3 Identifying threads

Every thread within a UNIX process has a unique identifier referred to as the *thread ID* or simply the TID. The TID is an integral value whose type is specified as `thread_t`. The TIDs can be reused within the same UNIX process. The function `thr_self()` can report the TID of the thread executing the command. The thread creating a new thread can get its TID by passing a `thread_t` type in the thread id parameter of `thr_create()`.

2.3.1 Syntax

```
#include <thread.h>
thread_t thr_self( void );
```

2.3.2 Sample Code

```
/* Rest of code */
printf(" The thread id of this thread is : %d\n", thr_self);
/* Rest of code */
```

2.4 Waiting for thread termination

A thread calls `thr_join()` function to wait for the normal termination of a sibling thread. The `thr_join()` function is similar to the `wait()` system call. When a thread requests to be joined with a sibling thread's execution stream, the requesting thread will wait for the sibling thread to terminate. After the sibling thread terminates, the execution streams are said to be joined and the calling thread resumes its execution.

NOTE *: Only threads created without the THR_DETACHED flag can be joined.*

2.4.1 Syntax

```
#include <thread.h>
int thr_join(
      thread_t    wait_for,          /* TID of thread to be joined */
      thread_t    *departed_thread,  /* TID of the joining thread */
      void        **status);         /* Status of terminating thread */
```

The calling thread can specify a particular thread to be joined by passing its TID as the `wait_for` parameter in the `thr_join()` function call. The calling thread will block until the two execution streams are joined. A zero `wait_for` value causes the `thr_join()` function to return upon the next joinable sibling thread to issue a `thr_exit()`. If the `wait_for` is a nonzero value, the `thr_join()` function returns *ESRCH* if the thread exists but is not joinable.

The terminating thread's identifier is returned in the location pointed to by the `departed_thread` parameter. If `*departed_thread` is not *NULL*, it points to a location that is set to the ID of the terminated thread if `thr_join()` returns successfully.

If status is not *NULL*, it points to a location that is set to the exit status of the terminated thread if `thr_join()` returns successfully.

Multiple threads cannot wait for the same thread to terminate. One thread will return successfully and the others will fail with a *ESRCH* return value.

2.4.2 Return Value

Zero is returned when successful. A non-zero value indicates an error.

2.4.3 Errors

If any of the following conditions are detected, *thr_join* fails and returns the corresponding value:

ESRCH	wait_for is not a valid, undetached thread in the current process.
EDEADLK	wait_for specifies the calling thread.

2.4.4 Sample Code

```
/* Rest of code */
thread_t Tid;
void *status = NULL;

thr_create( NULL, 0, function, arg, 0, &Tid);   /* Create a thread and get its
TID */
int error = thr_join( Tid, NULL, &status);      /* Wait for the thread to join
and get status */

if( (error == 0) && (status != NULL) )          /* Print message if successful
return */
printf("Thread %d has successfully joined\n", Tid);
/* Rest of code */
```

2.5 Terminating a thread

A UNIX thread can have voluntary or involuntary termination. A UNIX thread is considered to have voluntarily terminated by returning from the thread's start routine, or by calling the `thr_exit()` function.

<u>NOTE</u> : Although the two things are similar, it is considered good programming practice to end your threads by executing a `thr_exit()` function.

Involuntary thread termination is the result of a UNIX process terminating. Regardless of the UNIX process terminating by returning from `main()`, calling an explicit *exit* or as a result of an exception handling routine, the UNIX threads associated with the process are considered to be involuntarily terminated. If a thread spawns new threads and wants all of them to complete, then it should exit with a `thr_exit()` function call. This will ensure that the terminating thread waits for all non-daemon threads to finish before returning.

2.5.1 Syntax

```
#include <thread.h>
void thr_exit( void *status );
```

The distinction between voluntary and involuntary thread termination is significant with respect to the thread's termination `status`. When a non-detached thread voluntarily terminates, a sibling thread can examine the terminating thread's *status*. With involuntary termination, the thread's termination status has no significance.

All thread specific data bindings are deleted during the termination process. Default user level stacks are reclaimed by the UNIX threads library.

NOTE* : *If the exiting thread was created with a user defined stack, then it is the application's responsibility to free this memory or to reuse it.

In case of voluntary thread termination, the `thr_exit()` function transits the exiting thread from the *ON PROCESSOR* state to the *ZOMBIE* state. The UNIX thread library retains minimal information regarding the thread, such as TID and termination status value. If the terminating thread has a *DETACHED* attribute, then all information regarding the thread is released.

When a non-detached thread calls the `thr_exit ()`function, or terminates by returning from its `start_routine` and a sibling thread is waiting on its termination, then the sibling thread will complete its `thr_join()` call.

The exiting thread can specify an exit status by setting the parameter `status`. This status will then be returned to the thread calling the `thr_join()` function. A thread returning from the start routine without an explicit call to the `thr_exit()` function will have its exit status set to the value returned by the `start_routine`. The last non-detached thread to terminate causes the UNIX process to terminate. The exiting status of this thread is passed to the `exit()` function as the status value for the terminating UNIX process.

2.5.2 Sample Code

```
/* Rest of the code */
int *status;
if( error == 0)
      thr_exit( (void *) 0);         /* Return 0 as error status */
else
      thr_exit( (void *) status);    /* Return the status value */
/* Rest of the code */
```

2.6 Examining the concurrency level

The concurrency level of a UNIX process is determined by the number of kernel level LWPs in the multiplexing pool, plus the number of bound threads. Together, this represents the maximum number of threads having the potential to execute concurrently for a UNIX process. If there are additional multiplexing threads in the same UNIX process, these threads will remain in the *RUNNABLE* state until a multiplexing LWP from the pool is available.

The `thr_getconcurrency()` function call returns the minimum number of kernel LWPs in the multiplexing LWP pool. If the concurrency has not been set before, then this function returns a zero value to denote that the concurrency is being controlled automatically by the UNIX threads library. If the concurrency level has been explicitly adjusted before, then the function returns the current concurrency value.

2.6.1 Syntax

```
#include <thread.h>
int thr_getconcurrency( void );
```

2.6.2 Sample Code

```
/* Rest of the code */
printf("The current concurrency level is: %d\n", thr_getconcurrency() );
/* Rest of the code */
```

2.7 Setting the concurrency level

The `set_concurrency()` function adjusts the minimum concurrency level required by the application. This function registers a request and does not directly change the size of the multiplexing LWP pool. The `thr_setconcurrency()` function simply sets the minimum number of multiplexing LWPs requested by the application and returns from the function.

2.7.1 Syntax

```
#include <thread.h>
int thr_setconcurrency( int new_level );
```

The default or *automatic* concurrency control is used when `new_level` is given as zero. The default is one multiplexing LWP. Specifying a nonzero value for the concurrency level does not necessarily guarantee that the exact concurrency level will be realized. If the requested size would cause the number of LWPs for the user to exceed the system limit, then new LWPs will only be created only upto that limit. Alternatively, if the requested size is greater than the number of user-level threads, the size of the LWP pool will diminish over time to be less than or equal to the number of user-level threads.

On the other extreme, if the requested size would be below the number of user-level threads in the *RUNNABLE* state, then the UNIX threads library will lower the number of kernel-level LWPs in the multiplexing LWP pool to that value. This way the size of the multiplexing LWP pool can be lowered down to the size requested by the application.

2.7.2 Sample Code

```
void CheckConcurrency( void )
{
      int concurrency_level = 0;
      concurrency_level = thr_getconcurrency();
      ++concurrency_level;
      thr_setconcurrency( concurrency_level );
}
```

2.8 Suspending threads

The execution of a UNIX thread can be suspended and subsequently resumed by sibling threads of a UNIX process. There are two methods available for suspending a thread's potential for execution.

When the thread is first created , the *THR_SUSPEND* flag may be specified to immediately transition the new thread to the *suspended* state.

The other method is to explicitly call the `thr_suspend()` function to suspend the execution of the executing thread.

2.8.1 Syntax

```
#include <thread.h>
int thr_suspend( thread_t target_thread );
```

In using the `thr_suspend()` function the parameter `target_thread` is the thread identifier of the sibling thread to be suspended. The `thr_suspend()` function will not return control to the calling thread until the `target_thread` has successfully been suspended, or if an error condition occurs.

Some implementations of the UNIX threads interface use a notification scheme to alert the to alert the target thread of a pending suspension. The requesting thread does not directly suspend the target thread. Instead, it notifies the target thread that its suspension is pending. The requesting thread will then wait for the target thread to execute. When the target thread transitions to the *ON_PROCESSOR* state, it will see the pending suspension request and will release any critical implementation dependent resources it is holding. The target thread will suspend itself, and will transition from the *ON_PROCESSOR* state to the *SUSPENDED* state.

A multithreaded application can have several threads trying to suspend the same target thread. The implementation of the UNIX threads library permits only one of the requesting threads to send the suspension request to the target thread. The other threads will recognize the suspension request and will not issue any of their own. They will simply wait till the target thread goes to the *SUSPENDED* state. All requesting threads will be blocked till the suspension.

A bound thread is suspended through the kernel-level LWP function and not `thr_suspend()`. This is done since the LWP is not used for any other thread, and thus cannot perform any useful work when the thread is suspended. It transitions from the *ON_PROCESSOR* state to the *SUSPENDED* state.

If the target thread is a multiplexed thread, then the target thread must first transition to the *ON_PROCESSOR* state before it can be transitioned to the *SUSPENDED* state. Thus if the target thread is currently in the *SLEEP* state, it must first transition to the *RUNNABLE* state and then to the *ON_PROCESSOR* state before going to the *SUSPENDED* state. Unlike a bound thread, now the free LWP will be made available to other multiplexed threads.

2.8.2 Return Value

Zero is returned when successful. A non-zero value indicates an error.

2.8.3 Errors

If any of the following conditions are detected, `thr_suspend()` or `thr_continue()` fails and returns the corresponding value:

ESRCH	`target_thread` cannot be found in the current process.

2.9 Resuming suspended threads

The `thr_continue()` function resumes the execution of a suspended thread sibling thread.

2.9.1 Syntax

```
#include <thread.h>
int thr_continue( thread_t target_thread );
```

The parameter `target_thread` is the TID of the thread to be continued. If the target thread is not in the suspended state, then this function has no effect.

If the `target_thread` is a multiplexed thread then it will be transitioned from the *SUSPENDED* state to the *RUNNABLE* state.

If the `target_thread` is a bound thread then it will be transitioned from the *SUSPENDED* state directly to the *ON_PROCESSOR* state.

2.9.2 Return Value

Zero is returned when successful. A non-zero value indicates an error.

2.9.3 Errors

If any of the following conditions are detected, `thr_suspend()` or `thr_continue()` fails and returns the corresponding value:

ESRCH	`target_thread` cannot be found in the current process.

2.10 Yielding a thread's execution

A thread can yield its execution in favor of running another thread by calling the `thr_yield()` function.

2.10.1 Syntax

```
#include <thread.h>
void thr_yield( void );
```

If a multiplexed thread calls this function, then the multiplexing LWP is yielded in favor of other multiplexed threads in the *RUNNABLE* state.

If a bound thread calls this function then it yields the processor to another thread with equal or higher priority.

3. Biblography

1. Programming with UNIX threads - Charles J. Northrup.
2. Sun Web site - http://www.sun.com/developer-products/sig/threads
3. Sun Solaris thread programming manual.